# SMART CONTRACT AUDIT REPORT

for

# Venus MintBehalf

Prepared By: Yiqun Chen

PeckShield

June 14, 2021

## Document Properties

| | |
|---|---|
| Client | Venus |
| Title | Smart Contract Audit Report |
| Target | Venus MintBehalf |
| Version | 1.0 |
| Author | Shulin Bie |
| Auditors | Shulin Bie, Xuxian Jiang |
| Reviewed by | Yiqun Chen |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | June 14, 2021 | Shulin Bie | Release Candidate |

## Contact

For more information about this document and its contents, please contact PeckShield Inc.



| | |
|---|---|
| Name | Yiqun Chen |
| Phone | +86 183 5897 7782 |
| Email | contact@peckshield.com |

# Contents

# 1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Venus MintBehalf feature, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

## 1.1 About Venus MintBehalf

The Venus protocol is designed to enable a complete algorithmic money market protocol on Binance Smart Chain (BSC). Venus enables users to utilize their cryptocurrencies by supplying collateral to the protocol that may be borrowed by pledging over-collateralized cryptocurrencies. It also features a synthetic stablecoin (VAI) that is not backed by a basket of fiat currencies but by a basket of cryptocurrencies. Venus utilizes the BSC for fast, low-cost transactions while accessing a deep network of wrapped tokens and liquidity. The audited Venus MintBehalf support allows the user mints VTokens on behalf of others, which brings more flexibility for the Venus protocol.

The basic information of the Venus MintBehalf feature is as follows:

Table 1.1: Basic Information of Venus MintBehalf

| Item | Description |
|---|---|
| Name | Venus |
| Website | https://venus.io/ |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | June 14, 2021 |

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/VenusProtocol/venus-protocol/pull/51/commits/d4b53e0 (d4b53e0)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/VenusProtocol/venus-protocol/pull/51/commits/128fc1d (128fc1d)

## 1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

| Impact \ Likelihood | *High* | *Medium* | *Low* |
|---|---|---|---|
| *High* | Critical | High | Medium |
| *Medium* | High | Medium | Low |
| *Low* | Medium | Low | Low |

## 1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

| Category | Checklist Items |
|---|---|
| **Basic Coding Bugs** | Constructor Mismatch |
| | Ownership Takeover |
| | Redundant Fallback Function |
| | Overflows & Underflows |
| | Reentrancy |
| | Money-Giving Bug |
| | Blackhole |
| | Unauthorized Self-Destruct |
| | Revert DoS |
| | Unchecked External Call |
| | Gasless Send |
| | Send Instead Of Transfer |
| | Costly Loop |
| | (Unsafe) Use Of Untrusted Libraries |
| | (Unsafe) Use Of Predictable Variables |
| | Transaction Ordering Dependence |
| | Deprecated Uses |
| **Semantic Consistency Checks** | Semantic Consistency Checks |
| **Advanced DeFi Scrutiny** | Business Logics Review |
| | Functionality Checks |
| | Authentication Management |
| | Access Control & Authorization |
| | Oracle Security |
| | Digital Asset Escrow |
| | Kill-Switch Mechanism |
| | Operation Trails & Event Generation |
| | ERC20 Idiosyncrasies Handling |
| | Frontend-Contract Integration |
| | Deployment Consistency |
| | Holistic Risk Management |
| **Additional Recommendations** | Avoiding Use of Variadic Byte Array |
| | Using Fixed Compiler Version |
| | Making Visibility Level Explicit |
| | Making Type Inference Explicit |
| | Adhering To Function Declaration Strictly |
| | Following Other Best Practices |

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

| Category | Summary |
|---|---|
| **Configuration** | Weaknesses in this category are typically introduced during the configuration of the software. |
| **Data Processing Issues** | Weaknesses in this category are typically found in functionality that processes data. |
| **Numeric Errors** | Weaknesses in this category are related to improper calculation or conversion of numbers. |
| **Security Features** | Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.) |
| **Time and State** | Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads. |
| **Error Conditions, Return Values, Status Codes** | Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. |
| **Resource Management** | Weaknesses in this category are related to improper management of system resources. |
| **Behavioral Issues** | Weaknesses in this category are related to unexpected behaviors from code that an application uses. |
| **Business Logic** | Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. |
| **Initialization and Cleanup** | Weaknesses in this category occur in behaviors that are used for initialization and breakdown. |
| **Arguments and Parameters** | Weaknesses in this category are related to improper use of arguments or parameters within function calls. |
| **Expression Issues** | Weaknesses in this category are related to incorrectly written expressions within code. |
| **Coding Practices** | Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained. |

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

# 2 | Findings

## 2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Venus MintBehalf` support. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

| Severity | # of Findings | |
|---|---|---|
| Critical | 0 | |
| High | 0 | |
| Medium | 0 | |
| Low | 1 | ■ |
| Informational | 2 | ■ ■ |
| Total | 3 | |

We have previously audited the main Venus protocol. In this report, we exclusively focus on the specific pull request `d4b53e0`, we determine three issues that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussion of the issues are in Section 3.

## 2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key Venus MintBehalf Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | Low | Suggested Address Validity Check | Coding Practices | Fixed |
| PVE-002 | Informational | Non ERC20-Compliance Of VToken | Coding Practices | Confirmed |
| PVE-003 | Informational | Inconsistency Between Document And Implementation | Coding Practices | Fixed |

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

# 3 | Detailed Results

## 3.1 Suggested Address Validity Check

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VToken
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [4]

### Description

In the VToken contract, we notice there is a lack of parameter validity check in the mintBehalfFresh() function. To elaborate, we show below the related code snippet of this contract. We observe there is no address validity check for the second input argument receiver(line 588) inside the mintBehalfFresh () function. It may unnecessarily cause the loss of user's asset if the user accidently sets the receiver to address(0). It is suggested to apply a rigorous address validity check to avoid this specific case.

```
588     function mintBehalfFresh(address payer, address receiver, uint mintAmount) internal
            returns (uint, uint) {
589         /* Fail if mint not allowed */
590         uint allowed = comptroller.mintAllowed(address(this), receiver, mintAmount);
591         if (allowed != 0) {
592             return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                    MINT_COMPTROLLER_REJECTION, allowed), 0);
593         }
594
595         ...
596
597     }
```

Listing 3.1: VToken::mintBehalfFresh()

**Recommendation** Validate the input address receiver at the beginning of the mintBehalfFresh() function.

**Status** The issue has been addressed by the following commit: 128fc1d.

## 3.2 Non ERC20-Compliance Of VToken

- ID: PVE-002
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `VToken`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [3]

### Description

Each asset supported by the `Venus` protocol is integrated through a so-called `VToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `VTokens`, users can earn interest through the `VToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `VTokens` as collateral.

The implementation of `Venus MintBehalf` extends the original `VToken` contract and allows to mint `VTokens` on behalf of others. It also needs to follow the ERC20 standard. When analyzing this feature, we notice there is an ERC20-compliance issue in its implementation.

To elaborate, we show below the related code snippet of this contract. The ERC20 standard specifies that "a token contract which creates new tokens SHOULD trigger a `Transfer` event with the `_from` address set to `0x0` when tokens are created." [1] However, current `mintBehalfFresh()` logic emits the `Transfer` event by specifying the contract itself as the `_from`. For better ERC20 compliance, it is suggested to strictly follow the ERC20 standard.

```
    function mintBehalfFresh(address payer, address receiver, uint mintAmount) internal
        returns (uint, uint) {
        /* Fail if mint not allowed */
        uint allowed = comptroller.mintAllowed(address(this), receiver, mintAmount);
        if (allowed != 0) {
            return (failOpaque(Error.COMPTROLLER_REJECTION, FailureInfo.
                MINT_COMPTROLLER_REJECTION, allowed), 0);
        }

        ...

        /* We emit a MintBehalf event, and a Transfer event */
        emit MintBehalf(payer, receiver, vars.actualMintAmount, vars.mintTokens);
        emit Transfer(address(this), receiver, vars.mintTokens);

        /* We call the defense hook */
        comptroller.mintVerify(address(this), receiver, vars.actualMintAmount, vars.
            mintTokens);

        return (uint(Error.NO_ERROR), vars.actualMintAmount);
    }
```

Listing 3.2: `VToken::mintBehalfFresh()`

**Recommendation** Revise the `VToken` implementation to ensure its ERC20-compliance.

**Status** This issue has been confirmed. The team decides to leave it to keep consistency with the implementation of `mint` and reduce the risk of introducing bugs as a result of changing the behavior.

## 3.3 Inconsistency Between Document And Implementation

- ID: PVE-003
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `VToken`
- Category: Coding Practices [5]
- CWE subcategory: CWE-841 [2]

### Description

In the implementation of `MintBehalf` feature, we notice there is a misleading comment embedded among lines of the `mintBehalfInternal()` function, which brings unnecessary hurdles to understand and/or maintain the software.

To elaborate, we show below the related code snippet of this contract. The `mintBehalfInternal()` function is used to mint `VTokens` on behalf of the `receiver`. But we notice the comment (line 573) is "we still want to log the fact that an attempted **borrow** failed" when the `accrueInterest()` function returns an error. It will bring unnecessary hurdles to understand the function as this is related to mint, not borrow.

```
569    function mintBehalfInternal(address receiver, uint mintAmount) internal nonReentrant returns (uint, uint) {
570        uint error = accrueInterest();
571        if (error != uint(Error.NO_ERROR)) {
572            // accrueInterest emits logs on errors, but we still want to log the fact that an attempted mintBehalf failed
573            return (fail(Error(error), FailureInfo.MINT_ACCRUE_INTEREST_FAILED), 0);
574        }
575        // mintBelahfFresh emits the actual Mint event if successful and logs on errors, so we don't need to
576        return mintBehalfFresh(msg.sender, receiver, mintAmount);
577    }
```

Listing 3.3: `VToken::mintBehalfInternal()`

**Recommendation** Ensure the consistency between documents (including embedded comments) and implementation.

**Status** The issue has been addressed by the following commit: `128fc1d`.

# 4 Conclusion

In this audit, we have analyzed the Venus MintBehalf design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and synthetic stablecoins. The audited Venus MintBehalf support allows for minting VTokens on behalf of others. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

# References

[1] Fabian Vogelsteller And Vitalik Buterin. EIP-20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[2] MITRE. CWE-1068: Inconsistency Between Implementation and Documented Design. https://cwe.mitre.org/data/definitions/1068.html.

[3] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[4] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[8] PeckShield. PeckShield Inc. https://www.peckshield.com.